Test input generation for database programs using relational constraints

Marcozzi, Michaël; Vanhoof, Wim; Hainaut, Jean-Luc

Published in:
Proceedings of the Fifth International Workshop on Testing Database Systems

DOI:
10.1145/2304510.2304518

Publication date:
2012

Document Version
Peer reviewed version

Test Input Generation for Database Programs using Relational Constraints

Michaël Marcozzi*
Faculty of Computer Science
University of Namur
Rue Grandgagnage, 21
Namur, Belgium
mmr@info.fundp.ac.be

Wim Vanhoof
Faculty of Computer Science
University of Namur
Rue Grandgagnage, 21
Namur, Belgium
wva@info.fundp.ac.be

Jean-Luc Hainaut
Faculty of Computer Science
University of Namur
Rue Grandgagnage, 21
Namur, Belgium
jlh@info.fundp.ac.be

ABSTRACT

Databases are ubiquitous in software and testing of programs manipulating databases is thus essential to enhance the reliability of software. In this paper, we describe a clean and unified approach to automatically generate test inputs for such database programs. First, we propose a formal language, called ImperDB, to model database programs. ImperDB allows to model common program behaviors and data structures, as well as typical interaction scenarios between programs and databases. Secondly, we present a static analysis technique to generate test inputs for ImperDB programs, according to any chosen structural adequacy criterion. The technique considers an ImperDB program as a sequence of operations over a set of relational variables, modeling both the database original content and the program inputs. The problem of finding test inputs forcing the execution of a given path can then be transformed into the problem of solving constraints over the relational variables associated to the program. These constraints are expressed with the Alloy language and solved by the Alloy analyzer.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Code inspections and walk-throughs*; F.4.1 [**Mathematical Logic and formal languages**]: Mathematical Logic—*Logic and constraint programming*; H.2.0 [**Database Management**]: General—*Security, integrity, and protection*

General Terms

Reliability

Keywords

Structural software testing, Automatic test data generation, Database applications, Relational constraint solving, Alloy

*F.R.S.-FNRS Research Fellow

DBTest'12, May 21, 2012 Scottsdale, AZ, U.S.A.

1. INTRODUCTION

Testing [12] constitutes a significant approach to improve the reliability of software. Although testing cannot, in general, be used to establish formally correctness of a software artifact, the more the latter is tested with respect to different inputs, the more the confidence in its correctness is increased. The fact that a program typically exhibits a huge number of different behaviors to test makes the *automatic generation* of test inputs an interesting option if software is to be tested in an efficient and effective way. Two main challenges drive the work on automatic test input generation [21], namely how to define so-called adequacy criteria – i.e. measures that allow to assess the quality of a set of test inputs – and how to use these criteria in turn to drive the automatic generation of interesting test inputs for a given program.

In this work, we consider the automatic generation of test inputs for functional (unit) testing of *database-driven software* [21, 15]. Databases are nowadays ubiquitous in software and many programs interact intensively with a large, persistent and highly structured independent database. Functional testing of such database programs requires to assess the correct interaction between the program and the database.

In the field of software testing, most existing approaches to automatic test input generation consider a simplified model of program only [21]:

- A program implements a mathematical function from an input data domain to an output data domain.
- Input and output data domains, as well as the domains of the data types manipulated within a program are typically scalar domains like integers, booleans and reals, or simple combinations of scalar domains.
- Generating test inputs consists in selecting a relevant set of different input data to sufficiently exercise the properties of the tested program and/or of the mathematical function it implements with respect to some adequacy criterion.
- The chosen input data have no particular importance by themselves.

In the field of databases on the other hand, many existing approaches consider the generation of test databases as a standalone problem, independent of the data-flow in the programs interacting with the database under test [6]. Our

approach proposes to generate test inputs for a database program considering the full interaction between the tested database and the program. In particular, it offers a conceptually clean modeling that generalizes the simplified model of a program used in software testing, to account for the particularities of database programs:

- A database program computes the result of a mathematical function over the data it receives as input while it may perform at the same time frequent reads and writes into the independent database.
- The data read from and written into the database by the program may exhibit a complex structure, and must typically obey a large number of sometimes complex integrity constraints.
- Generating test inputs requires to select not only a relevant set of input data, but a relevant set of initial states of the database as well. As such, each test input consists in some input data coupled with an initial database state.
- The set of generated test inputs must allow not only to sufficiently exercise the properties of the tested program, but must also allow to assess that the program always modifies the initial database state in the expected way, without violating data integrity.

The contributions of our work can be summarized as follows. First, we propose a formal model for representing database programs as sequential programs interacting with a relational database. The language that we propose, called ImperDB, offers common imperative programming structures, some basic mechanisms for lists manipulation, as well as the common simple SQL interaction mechanisms between programs and databases. An important aspect of the language is that it allows for a clean modeling of all execution paths, including those that may lead to an erroneous interaction between the program and the database. Secondly, we generalize the white-box test input generation approach from [8] to ImperDB programs. Test inputs can be generated with respect to any (structural) code coverage criterion. The basic idea is to transform a (set of) execution path(s) that one wishes to exercise into a set of relational Alloy [10] constraints both on the program's input variables and on the input database of the program. Each solution to this set of constraints represents thus a test input and a minimal database content with respect to which the program can be executed and is guaranteed to follow the execution path associated to the constraints.

This paper is organized as follows: in Section 2 we present our formal model of database programs through a description of the syntax and semantics of the ImperDB language. Section 3 details the main mechanisms of our relational constraint-based approach to generate test inputs for ImperDB programs. Section 4 demonstrates how our technique can be applied to a sample ImperDB program. Finally, some conclusions and related work are provided in section 5.

2. IMPERDB: A FORMAL LANGUAGE FOR DATABASE PROGRAMS

Like any imperative language, ImperDB allows to write sequences of statements and to control the program flow using conditions and loops.

⟨*program*⟩ ::= ⟨*elem*⟩*

⟨*elem*⟩ ::= IF ⟨*cond*⟩ THEN ⟨*program*⟩ ELSE ⟨*program*⟩ ENDIF;
| WHILE ⟨*cond*⟩ DO ⟨*program*⟩ ENDWHILE;
| ⟨*statement*⟩;

An ImperDB program can process integer values, use integer lists and manipulate a database containing integer data. The exclusive use of integer values does not limit the expressive power of our model of database programs since all other usual primitive types such as booleans, strings, and floating point numbers, but also data structures such as arrays and matrices, can easily be mapped to integers and/or simulated using lists of integers. It does, however, make both the modeling and the use of a constraint solver conceptually simpler.

ImperDB is a strongly typed language which allows to define variables typed either as integer, as list of integers, or as integer table. The value of ImperDB variables can be set in three different contexts. First, *READ* statements assign an integer value from the outside world to one of the variables of the program. This can be used to model many different kinds of interaction between the program and the outside world, except from the interaction with the database: parameters received from a calling program, user prompt, network access, file read, etc. Variables initialized in this way are called *input variables* of the program. Secondly, *assignment* statements evaluate an integer or integer list expression and assign the obtained value to an *internal variable* of the program. Thirdly, *SQL SELECT* statements allow to assign the SQL table returned by a SQL query over the database to a table typed variable of the program. Program access to a table variable must be done row after row, using a cursor pointing at a single readable row. After reading from the database to a variable, a call to the *NEXT* statement allows to set the cursor in front of the first row of the table in the variable. Every subsequent *NEXT* statement will move the cursor one row ahead. If the *NEXT* statement is called when the cursor is in front of the last row, an exception is thrown within the program. *SQL INSERT/UPDATE/DELETE* statements allow the program to write data into the database. If the execution of a *SQL INSERT/UPDATE/DELETE* statement violates the database schema or integrity constraints, the database remains intact and an exception is thrown within the program. A *NEXT* or a *SQL INSERT/UPDATE/DELETE* statement can be wrapped into a *CATCH* statement. A *CATCH* statement will set an integer internal variable to 1 if an exception has been thrown by the wrapped statement, and to 0 otherwise. If an exception remains uncaught, the program immediately terminates.

⟨*var*⟩ ::= `[A-Z][-A-Z0-9]*`

⟨*statement*⟩ ::= READ(⟨*var*⟩) `(READ statement)`
| ⟨*var*⟩ = ⟨*expr*⟩ `(Assignment statement)`
| ⟨*var*⟩ = ⟨*db-read*⟩ `(SQL SELECT statement)`
| NEXT(⟨*var*⟩) `(NEXT statement)`
| ⟨*db-write*⟩ `(SQL INSERT/UPDATE/DELETE statement)`
| ⟨*var*⟩ = CATCH(NEXT(⟨*var*⟩)) `(CATCH statement)`
| ⟨*var*⟩ = CATCH(⟨*db-write*⟩) `(CATCH statement)`

ImperDB allows full arithmetics over integers and has the basic operations over lists of integers (concatenation of an element to a list and selecting the head, respectively, tail of a list). All expressions and statements in an ImperDB program are supposed to be well-typed, and lists are supposed

to be immutable objects. Lists can be used to simulate complex dynamic data structures such as arrays, sets, strings, matrices, etc.

⟨*expr*⟩ ::= ⟨*var*⟩ `(Integer/integers list typed variable)`
| ⟨*expr*⟩ ⟨*op*⟩ ⟨*expr*⟩ `(Integer arithmetics)`
| ⟨*var*⟩.HEAD `(Extracting the first element from the list in <var>)`
| ⟨*var*⟩(⟨*attr*⟩) `(Integer value of the attribute <attr> in the pointed row of the SQL table in <var>)`
| [0-9]+ `(Number)`
| (- ⟨*expr*⟩) `(Opposite value of an integer expression)`
| NIL `(Empty list)`
| (⟨*expr*⟩$_1$,⟨*expr*⟩$_2$) `(Appending integer <expr>`$_1$ `at the beginning of list <expr>`$_2$`)`
| ⟨*var*⟩.TAIL `(Removing the first element from the list in <var>)`

ImperDB allows all basic logic operations in if and while conditions. Every condition used in an ImperDB program should be built in a type-safe way.

⟨*cond*⟩ ::= (⟨*cond*⟩ | ⟨*cond*⟩) `(Logical OR)`
| (⟨*cond*⟩ & ⟨*cond*⟩) `(Logical AND)`
| ! (⟨*cond*⟩) `(Logical NOT)`
| TRUE `(Logical TRUE value)`
| FALSE `(Logical FALSE value)`
| (⟨*expr*⟩ == ⟨*expr*⟩) `(Integer equality)`
| (⟨*expr*⟩ < ⟨*expr*⟩) `(Integer inequality)`
| (⟨*var*⟩ == NIL) `(List emptiness)`

The database manipulated by an ImperDB program is specified by a relational schema, which can be extended with a set of simple integrity constraints. The schema describes a set of tables containing one or several mandatory integer attributes. For every table, a set of one or several attributes of the table must be declared primary key of the table. Some of the attributes of a table can be declared to constitute a foreign key that references another table in the schema. A row cannot be updated or deleted as long as there exists at least another row in the database that references it. Cycles in rows referencing are not allowed. Simple extra arithmetic constraints can be declared between the attributes of a table. An ImperDB program can read from and write into its associated database through simple static well-formed SQL statements SELECT, INSERT, UPDATE and DELETE. The result of such a statement over the database is fully predictable and deterministic. The whole interaction between the program and the database is executed within a single transaction.

⟨*db-read*⟩ ::= SELECT [⟨*attr*⟩,]*⟨*attr*⟩
FROM ⟨*rel*⟩
WHERE ⟨*db-cond*⟩

⟨*db-write*⟩ ::= ⟨*insert*⟩ | ⟨*update*⟩ | ⟨*delete*⟩

⟨*insert*⟩ ::= INSERT INTO ⟨*rel*⟩([⟨*attr*⟩,]*⟨*attr*⟩)
VALUES ([⟨*expr*⟩,]*⟨*expr*⟩)

⟨*update*⟩ ::= UPDATE ⟨*rel*⟩
SET [⟨*attr*⟩=⟨*db-expr*⟩,]*⟨*attr*⟩=⟨*db-expr*⟩
WHERE ⟨*db-cond*⟩

⟨*delete*⟩ ::= DELETE FROM ⟨*rel*⟩
WHERE ⟨*db-cond*⟩

⟨*attr*⟩ ::= [A-Za-z]+ `(Name of an attribute of a relation)`

⟨*rel*⟩ ::= [A-Za-z]+ `(Name of a relation)`

⟨*db-cond*⟩ ::= (⟨*db-cond*⟩ | ⟨*db-cond*⟩) `(Logical OR)`
| (⟨*db-cond*⟩ & ⟨*db-cond*⟩) `(Logical AND)`
| ! (⟨*db-cond*⟩) `(Logical NOT)`
| TRUE `(Logical TRUE value)`
| FALSE `(Logical FALSE value)`
| (⟨*attr*⟩ == ⟨*db-expr*⟩) `(Integer equality)`
| (⟨*attr*⟩ < ⟨*db-expr*⟩) `(Integer inequality)`

⟨*db-expr*⟩ ::= ⟨*var*⟩ `(Integer-typed variable)`
| ⟨*attr*⟩ `(Attribute value in the updated row)`
| ⟨*db-expr*⟩ ⟨*op*⟩ ⟨*db-expr*⟩ `(Integer arithmetics)`
| ⟨*var*⟩.HEAD `(Extracting first integer element from the list in <var>)`
| ⟨*var*⟩(⟨*attr*⟩) `(Integer value of the attribute <attr> in the current row of the SQL table in <var>)`
| [0-9]+ `(Number)`
| (- ⟨*db-expr*⟩) `(Opposite value of an integer expression)`

If we consider a correct ImperDB program together with the extended schema of the database it manipulates, then a *test input* for this system is a valuation for the program's input variables together with an instance of the database schema describing the state of the database before running the program.

3. TEST INPUT GENERATION USING RELATIONAL CONSTRAINTS

In so-called white-box, or structural testing [21], the adequacy of a set of test inputs is determined in terms of the amount of code that is covered by the set of test inputs. Many structural adequacy criteria have been proposed, including for example *statement coverage* and *branch coverage* that state, respectively, that every statement or branch in the program must be covered by the set of test inputs. A statement or branch is covered if there is at least one test input in the set that makes the execution reach the statement or branch.

In [8], a technique is proposed that allows to automatically generate test inputs for simple imperative programs that manipulate integer and list variables. The basic idea behind the approach is to transform any given path through the control-flow graph of the program into a set of constraints on the program input variables such that when the program is executed with respect to input values satisfying these constraints, the execution is guaranteed to follow the given path. The concrete input values are computed by a dedicated constraint solver. The method is independent of a particular coverage criterion since any structural coverage criterion can be accounted for by carefully building an appropriate finite set of paths through the control-flow graph and generating a test input for each such path.

In this work, we generalize the work of [8] to ImperDB programs. Like the original approach, our method is to some extent independent of, and can be parameterized by, the chosen adequacy criterion. In particular, it can be tailored to generate input data and input databases corresponding to execution paths where an uncaught exception is thrown within the program. Testing the existence of such paths is important, as they correspond to typical cases of erroneous interaction between the program and the database.

The core of the proposed method consists in modeling the execution of an ImperDB program as a sequence of successive simple operations on a set of input variables, describing mathematical relations over integers. The initial content of

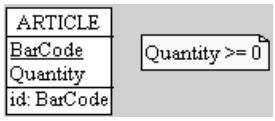

Figure 1: The stock database.

every table in the manipulated database can naturally be modeled as such a relational variable. Every integer input variable in the program can also be modeled as a relational variable, as an integer value can be seen as a singleton unary relation. Considering this model of program execution, the conditions over the input data and input database that force the execution to follow a given path of the program can be expressed as a set of constraints over the relational input variables of the program, in a similar way to [8]. The extended schema of the database manipulated by the tested program can also be modeled as constraints over the relational variables that represent the input database. The constraints modeling the execution path and the extended schema of the database can be expressed in a single set of constraints using the Alloy language [10] and solved using the Alloy analyzer [10]. Each solution to this set of constraints represents thus a test input, including key parts of the database, with respect to which the program can be executed and is guaranteed to follow the execution path associated to the constraints.

Alloy is a widely used declarative specification language. An Alloy specification is a collection of relational constraints that describes a set of structures to be discovered. The Alloy analyzer is a program which allows to solve the relational constraints in order to find structures that satisfy them. Basically, it transforms the set of relational constraints into an equivalent set of boolean constraints, and solves them using a SAT solver.

4. EXAMPLE

Let us consider a simple ImperDB program, used to update a stock database (Figure 1) upon replenishment of a given article. The database contains a single table 'Article' with two integer attributes: 'BarCode' (the primary key) and 'Quantity'. The value of the 'Quantity' attribute must be greater or equal to 0. The following program reads the bar code of the article as well as the quantity of newly arrived items for this article. If the article is already part of the database, the existing quantity is updated; otherwise, the article is added to the database. Afterwards, the program reads the total number of available items of the replenished article from the database and returns a list containing this value, as well as a flag indicating a risk of shortage for the article. A risk of shortage for a given article exists if the total quantity of this article in the stock is lower than 100.

```
read(BAR_CODE);
read(NUMBER_OF_NEW_ARTICLES);

CHECK_EMPTY =
select BarCode
from Article
where BarCode == BAR_CODE;

IS_EMPTY = catch(next(CHECK_EMPTY));

if (IS_EMPTY==1) then
  insert into Article(BarCode,Quantity)
  values (BAR_CODE,NUMBER_OF_NEW_ARTICLES);
else
  update Article
  set Quantity=Quantity+NUMBER_OF_NEW_ARTICLES
  where barcode == BAR_CODE;
endif;

QUANTITY_TABLE =
select Quantity
from Article
where BarCode == BAR_CODE;

next(QUANTITY_TABLE);

QUANTITY = QUANTITY_TABLE(Quantity);

if (QUANTITY < 100)
then
  RETURN = (QUANTITY,(1,NIL));
else
  RETURN = (QUANTITY,(0,NIL));
endif;
```

Let us show how a test input generator could automatically build an Alloy constraint model so that each possible solution to this model is a test input that allows to exercise a given execution path in the program. We consider the execution path where the THEN branches of both the first and second IF statements in the program are executed, and where neither the INSERT statement nor the NEXT statement throw an exception during execution.

The 'Article' table is modeled by a class 'Article' and two functional relations 'BarCode' and 'Quantity', mapping each object of type 'Article' to an integer. Two facts are added to the constraint model to express that 'BarCode' is the primary key of the 'Article' table, and that the 'Quantity' attribute cannot be lower than zero.

```
sig Article { BarCode : Int, Quantity : Int }

fact { all a,b:Article | (!(a = b) ⇔ !(a·BarCode = b·BarCode))}

fact { all a:Article | a·Quantity ≥ 0 }
```

Next, three Alloy variables are used to model the relational input variables of the program. The 'DBarticle0' variable is typed as a set of 'Article' objects and models as such the content of the 'Article' table at the start of the program. The 'INPBARCODE' and 'INPNUMBEROFNEWARTICLES' variables are typed as integers and they model, respectively, the BAR-CODE and NUMBER-OF-NEW-ARTICLES variables of the program.

```
sig DBarticle0 in Article {}

one sig INPBARCODE in Int {}

one sig INPNUMBEROFNEWARTICLES in Int {}
```

The behavior of the first SELECT statement in the program is modeled through an Alloy function f0. A fact is added to the constraint model to state that the Alloy variable v0 (modeling the program variable CHECK-EMPTY) contains the output of the function f0 applied to the initial content of the 'Article' table and to the value of the BAR-CODE variable.

```
fun f0[arts: set Article,bc: Int]:set Int{f1[arts,bc]·BarCode}

fun f1[arts: set Article,bc: Int]:set Article{(arts <:BarCode)·bc}

sig v0 in Int {}

fact { v0 = f0[DBarticle0,INPBARCODE] }
```

The constraint model should restrain the 'DBarticle0', 'INPBARCODE' and 'INPNUMBEROFNEWARTICLES' variables to the values that force the program to follow the THEN branch of the first IF statement. This occurs only if the first SELECT statement of the program returns an empty result. Hence a fact is added to the model which constrains the variable v0 to contain the empty set.

```
fact { #v0 = 0 }
```

Another fact is added to the constraint model to state that the Alloy variable DBarticle1 models the updated content of the 'Article' table after the successful execution of the INSERT statement.

```
sig DBarticle1 in Article {}

pred p0[artsAfter: set Article,bc: Int,nona: Int] {
one a: Article | (a in artsAfter) && (a·BarCode = bc)
                 && (a·Quantity = nona) && !(a in DBarticle0)
}

fact { p0[DBarticle1,INPBARCODE,
INPNUMBEROFNEWARTICLES] }
```

The second SELECT statement of the program is modeled in a similar way to the first one. It is applied to the updated content of the database, modeled by the variable DBarticle1.

```
fun f2[arts:set Article,bc:Int]:set Int{f3[arts,bc]·Quantity}

fun f3[arts:set Article,bc:Int]:set Article{(arts <:BarCode)·bc}

sig v1 in Int {}

fact { v1 = f2[DBarticle1,INPBARCODE] }
```

The model is furthermore constrained to reflect the fact that the NEXT(QUANTITY-TABLE) statement does not throw an exception and that the execution path covers the THEN branch of the second IF statement in the program. This means that the QUANTITY-TABLE table should contain one row whose attribute value is lower than 100.

```
one sig v1el1 in v1 {}
sig v1mel1 in v1 {}
fact { v1 − v1mel1 = v1el1 }
one sig v2 in Int {}
fact { v2 = v1el1}
fact { v2 < 100 }
```

Finally, a fact is added to account for the assignment of the RETURN variable. This constraint is not necessary to find correct input data for the considered execution path, but it allows showing list modeling in Alloy.

```
sig List { head: Int, tail: List + Nil }
one sig Nil {}
one sig v3 in List {}

fact {
v3·head = v2 && v3·tail·head = 1 && v3·tail·tail = Nil
}
```

Note that the above model can be derived automatically from the extended relational database schema and the program code. The Alloy analyzer can than be used to find a valuation for the relational variables which satisfies this constraint model by searching for a counterexample for the following assertion:

```
assert inputsDoNotExist {
!(DBarticle0 in Article && INPBARCODE in Int
&& INPNUMBEROFNEWARTICLES in Int) }

check inputsDoNotExist
```

The Alloy analyzer returns the following valuation for the three relational input variables of the program: {DBarticle0={}, INPBARCODE={6}, INPNUMBEROFNEWARTICLES={3}}. This corresponds to input data where the 'Article' table is empty, and three articles with bar code six should be added to the stock database. It is easy to see that these input data effectively exercise the considered execution path in the program.

5. CONCLUSION AND RELATED WORK

In this work, we present a language to model database programs, as well as a static analysis technique to generate test inputs for the modeled programs, according to any chosen white-box adequacy criterion. The technique consider database programs as sequences of operations over a set of initial relational variables, modeling both the database original state and the program inputs. The problem of finding inputs driving the execution of a given program path can then be transformed into the problem of solving a set of Alloy constraints over the initial relational variables of the program.

In future work, we intend to propose a complete formalization and evaluation of our test data generation technique and to generalize the ImperDB language to more complex SQL statements.

An early approach to have considered test data generation for imperative programs interacting with a relational SQL database is [2]. The paper proposes to transform the program, thereby inserting new variables representing the database structure, and translating all SQL statements and integrity checks into imperative program code. Classical white-box testing approaches can then be applied to the modified program. In [9], the authors propose an algorithm for testing an imperative program performing SELECT queries on a relational SQL database, based on a simultaneous concrete and symbolic (concolic) dynamic exploration of some or all of its execution paths. Concolic execution runs the program on random input data and on a randomly populated input database. Given the dynamic exploration of an execution path of the program, the authors model and solve the problem of finding other inputs, allowing to explore dynamically another execution path, as a set of integer and string constraints over the quantity and field contents of the records in the database and over the input variables of the tested program. These constraints must be combined with the constraints derived from the database schema.

Compared to all of these approaches, our approach does not need to transform the original program, offers a clean modeling of the problem as a single relational constraints

system, and allows to account for INSERT/UPDATE/DELETE statements that are commonly used in database applications. On the other hand, our approach only considers static SQL where the concolic approaches allow to account for dynamic SQL. Investigating whether and how dynamic SQL can be integrated with our approach, possibly relying on static analysis [18, 16], is a topic for further research.

In [3, 4], test database states are treated independently of the program control flow, and generated, with the help of the user, on the basis of the database schema and of heuristics aiding at the generation of states likely to expose program faults. Similar approaches have been proposed [14, 20, 1, 13, 17, 7, 19] that all generate database instances independently of the program's data- and control flow.

Finally, the translation between database schemas/applications and Alloy models has already been considered in other contexts [5, 11].

6. ACKNOWLEDGMENTS

This work has been funded by the Belgian Fund for Scientific Research (F.R.S.-FNRS). The authors would like to thank Pierre-Yves Schobbens for useful discussions and the anonymous reviewers for their valuable comments.

7. REFERENCES

[1] C. Binnig, D. Kossmann, and E. Lo. Reverse query processing. In *in the IDEA System. Int. Symp. on Advanced Database Technologies and Their Integration*, 1994.

[2] M. Y. Chan and S. C. Cheung. Testing database applications with sql semantics. In *In Proceedings of the 2nd International Symposium on Cooperative Database Systems for Advanced Applications*, pages 363–374. Springer, 1999.

[3] D. Chays, Y. Deng, P. G. Frankl, S. Dan, F. I. Vokolos, and E. J. Weyuker. An agenda for testing relational database applications: Research articles. *Softw. Test. Verif. Reliab.*, 14(1):17–44, Mar. 2004.

[4] D. Chays, J. Shahid, and P. G. Frankl. Query-based test generation for database applications. In *Proceedings of the 1st international workshop on Testing database systems*, DBTest '08, pages 6:1–6:6, New York, NY, USA, 2008. ACM.

[5] A. Cunha and H. Pacheco. Mapping between alloy specifications and database implementations. In *Proceedings of the 2009 Seventh IEEE International Conference on Software Engineering and Formal Methods*, SEFM '09, pages 285–294, Washington, DC, USA, 2009. IEEE Computer Society.

[6] C. de la Riva, M. J. Suárez-Cabal, and J. Tuya. Constraint-based test database generation for sql queries. In *Proceedings of the 5th Workshop on Automation of Software Test*, AST '10, pages 67–74, New York, NY, USA, 2010. ACM.

[7] C. de la Riva, M. J. Suárez-Cabal, and J. Tuya. Constraint-based test database generation for sql queries. In *Proceedings of the 5th Workshop on Automation of Software Test*, AST '10, pages 67–74, New York, NY, USA, 2010. ACM.

[8] F. Degrave, T. Schrijvers, and W. Vanhoof. Towards a framework for constraint-based test case generation. In *Proceedings of the 19th international conference on Logic-Based Program Synthesis and Transformation*, LOPSTR'09, pages 128–142, Berlin, Heidelberg, 2010. Springer-Verlag.

[9] M. Emmi, R. Majumdar, and K. Sen. Dynamic test input generation for database applications. In *Proceedings of the 2007 international symposium on Software testing and analysis*, ISSTA '07, pages 151–162, New York, NY, USA, 2007. ACM.

[10] D. Jackson. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, 2006.

[11] D. Jackson and M. Vaziri. Finding bugs with a constraint solver. *SIGSOFT Softw. Eng. Notes*, 25(5):14–25, Aug. 2000.

[12] C. Kaner, H. Q. Nguyen, and J. L. Falk. *Testing Computer Software*. John Wiley & Sons, Inc., New York, NY, USA, 2nd edition, 1993.

[13] S. A. Khalek, B. Elkarablieh, Y. O. Laleye, and S. Khurshid. Query-aware test generation using a relational constraint solver. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, ASE '08, pages 238–247, Washington, DC, USA, 2008. IEEE Computer Society.

[14] A. Neufeld, G. Moerkotte, and P. C. Lockemann. Generating consistent test data for a variable set of general consistency constraints. *VLDB J.*, 2(2):173–213, 1993.

[15] K. Pan, X. Wu, and T. Xie. Generating program inputs for database application testing. In *Proc. 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, November 2011.

[16] S. Thomas, L. Williams, and T. Xie. On automated prepared statement generation to remove sql injection vulnerabilities. *Inf. Softw. Technol.*, 51(3):589–598, Mar. 2009.

[17] M. Veanes, P. Grigorenko, P. Halleux, and N. Tillmann. Symbolic query exploration. In *Proceedings of the 11th International Conference on Formal Engineering Methods: Formal Methods and Software Engineering*, ICFEM '09, pages 49–68, Berlin, Heidelberg, 2009. Springer-Verlag.

[18] G. Wassermann, C. Gould, Z. Su, and P. Devanbu. Static checking of dynamically generated queries in database applications. *ACM Trans. Softw. Eng. Methodol.*, 16(4), Sept. 2007.

[19] D. Willmor and S. M. Embury. An intensional approach to the specification of test cases for database applications. In *Proceedings of the 28th international conference on Software engineering*, ICSE '06, pages 102–111, New York, NY, USA, 2006. ACM.

[20] J. Zhang, C. Xu, and S. C. Cheung. Automatic generation of database instances for white-box testing. In *Proceedings of the 25th International Computer Software and Applications Conference on Invigorating Software Development*, COMPSAC '01, pages 161–165, Washington, DC, USA, 2001. IEEE Computer Society.

[21] H. Zhu, P. A. V. Hall, and J. H. R. May. Software unit test coverage and adequacy. *ACM Comput. Surv.*, 29(4):366–427, Dec. 1997.